Parallelization Overheads

Marc Moreno Maza

University of Western Ontario, Canada

SHARCNET Summer Seminar, London
July 12, 2012
What is this tutorial about?

Why is my parallel program not reaching linear speedup? not scaling?
What is this tutorial about?

Why is my parallel program not reaching linear speedup? not scaling?

- The algorithm could lack of parallelism . . .
What is this tutorial about?

Why is my parallel program not reaching linear speedup? not scaling?

- The algorithm could lack of parallelism . . .
- The architecture could suffer from limitations . . .
Why is my parallel program not reaching linear speedup? not scaling?

- The algorithm could lack of parallelism . . .
- The architecture could suffer from limitations . . .
- The program does not expose enough parallelism . . .
What is this tutorial about?

Why is my parallel program not reaching linear speedup? not scaling?
- The algorithm could lack of parallelism . . .
- The architecture could suffer from limitations . . .
- The program does not expose enough parallelism . . .
- Or the concurrency platform suffers from overheads!
Why architectures will we discuss?

From multi to many cores
- Multicore architectures have brought parallelism to the masses.
Why architectures will we discuss?

From multi to many cores

- Multicore architectures have brought parallelism to the masses.
- However, multithreaded programs are not always as efficient as expected.
Why architectures will we discuss?

From multi to many cores

- Multicore architectures have brought parallelism to the masses.
- However, multithreaded programs are not always as efficient as expected.
- In fact, certain operations are not well suited for parallelization on multicores.
Why architectures will we discuss?

From multi to many cores
- Multicore architectures have brought parallelism to the masses.
- However, multithreaded programs are not always as efficient as expected.
- In fact, certain operations are not well suited for parallelization on multicores.
- GPUs have brought supercomputing to the masses.
Why architectures will we discuss?

From multi to many cores

- Multicore architectures have brought parallelism to the masses.
- However, multithreaded programs are not always as efficient as expected.
- In fact, certain operations are not well suited for parallelization on multicores.
- GPUs have brought supercomputing to the masses.
- GPUs are harder to program than multicores, though.
Why architectures will we discuss?

From multi to many cores

- Multicore architectures have brought parallelism to the masses.
- However, multithreaded programs are not always as efficient as expected.
- In fact, certain operations are not well suited for parallelization on multicores.
- GPUs have brought supercomputing to the masses.
- GPUs are harder to program than multicores, though.
- But they can provide better performances in terms of burdened parallelism.
What are the prerequisites?

- Some familiarity with algorithms and their analysis.
- Pascal Triangle, Euclidean Algorithm.
- Ideas about multithreaded programming.
- Some ideas about GPUs.
- Visit http://uwo.sharcnet.ca/
What are the objectives of this tutorial?

1. Understand that concurrent execution has a cost which cannot be captured by standard analysis of algorithms.
What are the objectives of this tutorial?

1. Understand that concurrent execution has a cost which cannot be captured by standard analysis of algorithms.
2. This cost is a separate issue from cache complexity, memory traffic, inappropriate thresholds, etc.
What are the objectives of this tutorial?

1. Understand that concurrent execution has a cost which cannot be captured by standard analysis of algorithms.
2. This cost is a separate issue from cache complexity, memory traffic, inappropriate thresholds, etc.
3. This cost essentially corresponds to thread (and thread block) scheduling, task migration (multi-cores), kernel launches (GPUs), etc.
What are the objectives of this tutorial?

1. Understand that concurrent execution has a cost which cannot be captured by standard analysis of algorithms.

2. This cost is a separate issue from cache complexity, memory traffic, inappropriate thresholds, etc.

3. This cost essentially corresponds to thread (and thread block) scheduling, task migration (multi-cores), kernel launches (GPUs), etc.

4. We call this cost *parallelization overheads*.
What are the objectives of this tutorial?

1. Understand that concurrent execution has a cost which cannot be captured by standard analysis of algorithms.
2. This cost is a separate issue from cache complexity, memory traffic, inappropriate thresholds, etc.
3. This cost essentially corresponds to thread (and thread block) scheduling, task migration (multi-cores), kennel launches (GPUs), etc.
4. We call this cost parallelization overheads.
5. We will see that parallelization overheads can be measured, and even analyzed theoretically.
What are the objectives of this tutorial?

1. Understand that concurrent execution has a cost which cannot be captured by standard analysis of algorithms.

2. This cost is a separate issue from cache complexity, memory traffic, inappropriate thresholds, etc.

3. This cost essentially corresponds to thread (and thread block) scheduling, task migration (multi-cores), kennel launches (GPUs), etc.

4. We call this cost **parallelization overheads**.

5. We will see that parallelization overheads can be measured, and even analyzed theoretically.

6. This can improve performances in a significant manner.
What are the objectives of this tutorial?

1. Understand that concurrent execution has a cost which cannot be captured by standard analysis of algorithms.
2. This cost is a separate issue from cache complexity, memory traffic, inappropriate thresholds, etc.
3. This cost essentially corresponds to thread (and thread block) scheduling, task migration (multi-cores), kernel launches (GPUs), etc.
4. We call this cost **parallelization overheads**.
5. We will see that parallelization overheads can be measured, and even analyzed theoretically.
6. This can improve performances in a significant manner.
7. In some cases, this can help discovering that a concurrency platform cannot support an algorithm.
Acknowledgments and references

Acknowledgments.

- Charles E. Leiserson (MIT), Matteo Frigo (Axis Semiconductor) Saman P. Amarasinghe (MIT) and Cyril Zeller (NVIDIA) for sharing with me the sources of their course notes and other documents.
- Yuzhen Xie (Maplesoft) and Anisul Sardar Haque (UWO) for their great help in the preparation of this tutorial.

References.

- *The Cilkview scalability analyzer* by Yuxiong He, Charles E. Leiserson and William M. Leiserson.
- http://www.csd.uwo.ca/~moreno/HPC-Resources.html
Plan

1. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join model and its implementation in Cilk
   - Detecting parallelization overheads and other issues
   - Anticipating parallelization overheads

2. GPU programming
   - The CUDA programming and memory models
   - Some principles for performance optimization
   - Anticipating parallelization overheads: plain polynomial division
   - Anticipating parallelization overheads: Euclidean Algorithm
   - An interlude: plain polynomial multiplication
   - Benchmarks and conclusions
Plan

1. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join model and its implementation in Cilk
   - Detecting parallelization overheads and other issues
   - Anticipating parallelization overheads

2. GPU programming
   - The CUDA programming and memory models
   - Some principles for performance optimization
   - Anticipating parallelization overheads: plain polynomial division
   - Anticipating parallelization overheads: Euclidean Algorithm
   - An interlude: plain polynomial multiplication
   - Benchmarks and conclusions
A multi-core processor is an integrated circuit to which two or more individual processors (called cores in this sense) have been attached.
Cores on a multi-core device can be **coupled tightly or loosely**:
- may share or may not share a cache,
- implement inter-core communications methods or message passing.

Cores on a multi-core implement the **same architecture features as single-core systems** such as instruction pipeline parallelism (ILP), vector-processing, hyper-threading, etc.
Cache Coherence (1/6)

**Figure:** Processor $P_1$ reads $x=3$ first from the backing store (higher-level memory)
Figure: Next, Processor $P_2$ loads $x=3$ from the same memory
Cache Coherence (3/6)

Figure: Processor $P_4$ loads $x=3$ from the same memory
Figure: Processor $P_2$ issues a write $x=5$
Cache Coherence (5/6)

Figure: Processor $P_2$ writes $x=5$ in his local cache
Cache Coherence (6/6)

Figure: Processor $P_1$ issues a read $x$, which is now invalid in its cache.
Multi-core processor (cntd)

**Advantages:**
- Cache coherency circuitry operate at higher rate than off-chip.
- Reduced power consumption for a dual core vs two coupled single-core processors (better quality communication signals, cache can be shared)

**Challenges:**
- Adjustments to existing software (including OS) are required to maximize performance
- Production yields down (an Intel quad-core is in fact a double dual-core)
- Two processing cores sharing the same bus and memory bandwidth may limit performances
- High levels of false or true sharing and synchronization can easily overwhelm the advantage of parallelism
Plan

1 Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join model and its implementation in Cilk
   - Detecting parallelization overheads and other issues
   - Anticipating parallelization overheads

2 GPU programming
   - The CUDA programming and memory models
   - Some principles for performance optimization
   - Anticipating parallelization overheads: plain polynomial division
   - Anticipating parallelization overheads: Euclidean Algorithm
   - An interlude: plain polynomial multiplication
   - Benchmarks and conclusions
From Cilk to Cilk++ and Cilk Plus

- Cilk has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo.
- Besides being used for research and teaching, Cilk was the system used to code the three world-class chess programs: Tech, Socrates, and Cilkchess.
- Over the years, the implementations of Cilk have run on computers ranging from networks of Linux laptops to an 1824-nodes Intel Paragon.
- From 2007 to 2009 Cilk has lead to Cilk++, developed by Cilk Arts, an MIT spin-off, which was acquired by Intel in July 2009 and became Cilk Plus, see http://www.cilk.com/
- Cilk++ can be freely downloaded at http://software.intel.com/en-us/articles/download-intel-cilk-sdk/
- Cilk is still developed at MIT http://supertech.csail.mit.edu/cilk/
Cilk++ (and Cilk Plus)

- Cilk++ (resp. Cilk) is a small set of linguistic extensions to C++ (resp. C) supporting fork-join parallelism.

- Both Cilk and Cilk++ feature a provably efficient work-stealing scheduler.

- Cilk++ provides a hyperobject library for parallelizing code with global variables and performing reduction for data aggregation.

- Cilk++ includes the Cilkscreen race detector and the Cilkview performance analyzer.
The named child function `cilk_spawn fib(n-1)` may execute in parallel with its parent.

Cilk++ keywords `cilk_spawn` and `cilk_sync` grant permissions for parallel execution. They do not command parallel execution.
A **scheduler**’s job is to map a computation to particular processors. Such a mapping is called a **schedule**.

- If decisions are made at runtime, the scheduler is **online**, otherwise, it is **offline**
- Cilk++’s scheduler maps strands onto processors dynamically at runtime.
The Cilk++ Platform

int fib(int n) {
    if (n<2) return (n);
    else {
        int x,y;
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
        cilk_sync;
        return (x+y);
    }
}

Cilk++ source

Cilk++ Compiler

Hyperobject Library

Hyperobject Analyzer

Cilkview Scalability Analyzer

Cilkscreen Race Detector

Runtime System

Parallel Regression Tests

Reliable Multi-Threaded Code

Exceptional Performance

Reliable Single-Threaded Code

Conventional Regression Tests

Serialization

Conventional Compiler

Linker

Binary

 Conventionally

Reliable Single-Threaded Code

Exceptional Performance

Reliable Multi-Threaded Code
Benchmarks for the parallel version of the cache-oblivious mm

Multiplying a 4000x8000 matrix by a 8000x4000 matrix

- on 32 cores = 8 sockets x 4 cores (Quad Core AMD Opteron 8354) per socket.
- The 32 cores share a L3 32-way set-associative cache of 2 Mbytes.

<table>
<thead>
<tr>
<th>#core</th>
<th>Elision (s)</th>
<th>Parallel (s)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>420.906</td>
<td>51.365</td>
<td>8.19</td>
</tr>
<tr>
<td>16</td>
<td>432.419</td>
<td>25.845</td>
<td>16.73</td>
</tr>
<tr>
<td>24</td>
<td>413.681</td>
<td>17.361</td>
<td>23.83</td>
</tr>
<tr>
<td>32</td>
<td>389.300</td>
<td>13.051</td>
<td>29.83</td>
</tr>
</tbody>
</table>
So does the (tuned) cache-oblivious matrix multiplication
Plan

1. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join model and its implementation in Cilk
   - Detecting parallelization overheads and other issues
   - Anticipating parallelization overheads

2. GPU programming
   - The CUDA programming and memory models
   - Some principles for performance optimization
   - Anticipating parallelization overheads: plain polynomial division
   - Anticipating parallelization overheads: Euclidean Algorithm
   - An interlude: plain polynomial multiplication
   - Benchmarks and conclusions
The fork-join parallelism model

int fib (int n) {
    if (n<2) return (n);
    else {
        int x,y;
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
        cilk_sync;
        return (x+y);
    }
}

Example:
fib(4)

"Processor oblivious"

The computation dag unfolds dynamically.

We shall also call this model multithreaded parallelism.
The fork-join parallelism model

$T_p$ is the minimum running time on $p$ processors.

$T_1$ is the sum of the number of instructions at each vertex in the DAG, called the **work**.

$T_\infty$ is the minimum running time with infinitely many processors, called the **span**. This is the length of a path of maximum length from the root to a leaf.

$T_1 / T_\infty$: **Parallelism**.

- *Work law*: $T_p \geq T_1 / p$.
- *Span law*: $T_p \geq T_\infty$.

**Figure**: Instruction stream DAG.
A parallel program execution can have:

- **linear speedup**: $T_1 / T_P = \Theta(p)$
- **superlinear speedup**: $T_1 / T_P = \omega(p)$ (not possible in this model, though it is possible in others)
- **sublinear speedup**: $T_1 / T_P = o(p)$
For loop parallelism in Cilk++

cilk_for (int i=1; i<n; ++i) {
    for (int j=0; j<i; ++j) {
        double temp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = temp;
    }
}

The iterations of a cilk_for loop execute in parallel.
Implementation of for loops in Cilk++

Up to details (next week!) the previous loop is compiled as follows, using a divide-and-conquer implementation:

```c
void recur(int lo, int hi) {
    if (hi > lo) { // coarsen
        int mid = lo + (hi - lo)/2;
        cilk_spawn recur(lo, mid);
        recur(mid+1, hi);
        cilk_sync;
    } else
        for (int j=0; j<hi; ++j) {
            double temp = A[hi][j];
            A[hi][j] = A[j][hi];
            A[j][hi] = temp;
        }
}
```
Analysis of parallel for loops

Here we do not assume that each strand runs in unit time.

- **Span of loop control**: $\Theta(\log(n))$
- **Max span of an iteration**: $\Theta(n)$
- **Span**: $\Theta(n)$
- **Work**: $\Theta(n^2)$
- **Parallelism**: $\Theta(n)$
For loops in the fork-join parallelism model: another example

cilk_for (int i = 1; i <= 8; i ++){
    f(i);
}

A cilk_for loop executes recursively as 2 for loops of \( n/2 \) iterations, adding a span of \( \Theta(\log(n)) \).

Figure: DAG for a cilk_for with 8 iterations.
The work-stealing scheduler (1/11)
The work-stealing scheduler (3/11)
The work-stealing scheduler (4/11)
The work-stealing scheduler (5/11)
The work-stealing scheduler (6/11)
The work-stealing scheduler (7/11)
The work-stealing scheduler (8/11)
The work-stealing scheduler (9/11)
The work-stealing scheduler (10/11)
The work-stealing scheduler (11/11)
Performances of the work-stealing scheduler

Assume that

- each strand executes in unit time,
- for almost all “parallel steps” there are at least $p$ strands to run,
- each processor is either working or stealing.

Then, the randomized work-stealing scheduler is expected to run in

$$T_P = T_1/p + O(T_\infty)$$
Overheads and burden

Many factors (simplification assumptions of the fork-join parallelism model, architecture limitation, costs of executing the parallel constructs, overheads of scheduling) will make $T_p$ larger in practice than $T_1/p + T_\infty$.

One may want to estimate the impact of those factors:
1. by improving the estimate of the randomized work-stealing complexity result
2. by comparing a Cilk++ program with its C++ elision
3. by estimating the costs of spawning and synchronizing

Cilk++ estimates $T_p$ as $T_p = T_1/p + 1.7$ burden.span, where burden.span is 15000 instructions times the number of continuation edges along the critical path.
Cilkview computes work and span to derive upper bounds on parallel performance.

Cilkview also estimates scheduling overhead to compute a burdened span for lower bounds.
The cilkview example from the documentation

Using cilk_for to perform operations over an array in parallel:

```c
static const int COUNT = 4;
static const int ITERATION = 1000000;
long arr[COUNT];
long do_work(long k){
    long x = 15;
    static const int nn = 87;
    for (long i = 1; i < nn; ++i)
        x = x / i + k % i;
    return x;
}
int cilk_main(){
    for (int j = 0; j < ITERATION; j++)
        cilk_for (int i = 0; i < COUNT; i++)
            arr[i] += do_work( j * i + i + j);
}```
1) Parallelism Profile

Work : 6,480,801,250 ins
Span : 2,116,801,250 ins
Burdened span : 31,920,801,250 ins
Parallelism : 3.06
Burdened parallelism : 0.20
Number of spawns/syncs: 3,000,000
Average instructions / strand : 720
Strands along span : 4,000,001
Average instructions / strand on span : 529

2) Speedup Estimate

2 processors: 0.21 - 2.00
4 processors: 0.15 - 3.06
8 processors: 0.13 - 3.06
16 processors: 0.13 - 3.06
32 processors: 0.12 - 3.06
A simple fix

Inverting the two for loops

```c
int cilk_main()
{
    cilk_for (int i = 0; i < COUNT; i++)
        for (int j = 0; j < ITERATION; j++)
            arr[i] += do_work( j * i + i + j);
}
```
1) Parallelism Profile

Work : 5,295,801,529 ins
Span : 1,326,801,107 ins
Burdened span : 1,326,830,911 ins
Parallelism : 3.99
Burdened parallelism : 3.99
Number of spawns/syncs: 3
Average instructions / strand : 529,580,152
Strands along span : 5
Average instructions / strand on span: 265,360,221

2) Speedup Estimate

2 processors: 1.40 - 2.00
4 processors: 1.76 - 3.99
8 processors: 2.01 - 3.99
16 processors: 2.17 - 3.99
32 processors: 2.25 - 3.99
### Timing

<table>
<thead>
<tr>
<th>version</th>
<th>#cores = 1</th>
<th>#cores = 2</th>
<th>#cores = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>timing(s)</td>
<td>timing(s)</td>
<td>timing(s)</td>
</tr>
<tr>
<td>original</td>
<td>7.719</td>
<td>9.611</td>
<td>10.758</td>
</tr>
<tr>
<td>improved</td>
<td>7.471</td>
<td>3.724</td>
<td>1.888</td>
</tr>
<tr>
<td></td>
<td>0.803</td>
<td>2.006</td>
<td>3.957</td>
</tr>
</tbody>
</table>
Plan

1. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join model and its implementation in Cilk
   - Detecting parallelization overheads and other issues
   - Anticipating parallelization overheads

2. GPU programming
   - The CUDA programming and memory models
   - Some principles for performance optimization
   - Anticipating parallelization overheads: plain polynomial division
   - Anticipating parallelization overheads: Euclidean Algorithm
   - An interlude: plain polynomial multiplication
   - Benchmarks and conclusions
Example 1: a small loop with grain size = 1

Code:

```c
const int N = 100 * 1000 * 1000;

void cilk_for_grainsize_1()
{
    #pragma cilk_grainsize = 1
    cilk_for (int i = 0; i < N; ++i)
        fib(2);
}
```

Expectations:
- Parallelism should be large, perhaps $\Theta(N)$ or $\Theta(N/\log N)$.
- We should see great speedup.
Speedup is indeed great...
but performance is lousy
Recall how `cilk_for` is implemented

**Source:**

```c
  cilk_for (int i = A; i < B; ++i)
    BODY(i)
```

**Implementation:**

```c
void recur(int lo, int hi) {
  if ((hi - lo) > GRAINSIZE) {
    int mid = lo + (hi - lo) / 2;
    cilk_spawn recur(lo, mid);
    cilk_spawn recur(mid, hi);
  } else
    for (int i = lo; i < hi; ++i)
      BODY(i);
}

recur(A, B);
```
### Default grain size

Cilk++ chooses a grain size if you don’t specify one.

```cpp
void cilk_for_default_grainsize()
{
    cilk_for (int i = 0; i < N; ++i)
    fib(2);
}
```

Cilk++’s heuristic for the grain size:

\[
\text{grain size} = \min \left\{ \frac{N}{8P}, 512 \right\}
\]

- Generates about \(8P\) parallel leaves.
- Works well if the loop iterations are not too unbalanced.
Speedup with default grain size
Large grain size

A large grain size should be even faster, right?

```c
void cilk_for_large_grainsize()
{
    #pragma cilk_grainsize = N
    cilk_for (int i = 0; i < N; ++i)
        fib(2);
}
```

Actually, no (except for noise):

<table>
<thead>
<tr>
<th>Grain size</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8.55 s</td>
</tr>
<tr>
<td>default (= 512)</td>
<td>2.44 s</td>
</tr>
<tr>
<td>$N$ (= $10^8$)</td>
<td>2.42 s</td>
</tr>
</tbody>
</table>
Speedup with grain size $= N$
Trade-off between grain size and parallelism

Use Cilkview to understand the trade-off:

<table>
<thead>
<tr>
<th>Grain size</th>
<th>Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6,951,154</td>
</tr>
<tr>
<td>default (= 512)</td>
<td>248,784</td>
</tr>
<tr>
<td>( N (= 10^8) )</td>
<td>1</td>
</tr>
</tbody>
</table>

In Cilkview, \( P = 1 \):

\[
\text{default grain size} = \min \left\{ \frac{N}{8P}, 512 \right\} = \min \left\{ \frac{N}{8}, 512 \right\}.
\]
Lessons learned

- Measure overhead before measuring speedup.
  - Compare 1-processor Cilk++ versus serial code.
- Small grain size $\Rightarrow$ higher work overhead.
- Large grain size $\Rightarrow$ less parallelism.
- The default grain size is designed for small loops that are reasonably balanced.
  - You may want to use a smaller grain size for unbalanced loops or loops with large bodies.
- Use Cilkview to measure the parallelism of your program.
Example 2: A for loop that spawns

Code:

```c
const int N = 10 * 1000 * 1000;

/* empty test function */
void f() { }

void for_spawn()
{
    for (int i = 0; i < N; ++i)
        cilk_spawn f();
}
```

Expectations:
- I am spawning \( N \) parallel things.
- Parallelism should be \( \Theta(N) \), right?
“Speedup” of for_spawn()
Insufficient parallelism

PPA analysis:
- PPA says that both work and span are $\Theta(N)$.
- Parallelism is $\approx 1.62$, independent of $N$.
- Too little parallelism: no speedup.

Why is the span $\Theta(N)$?

```c
for (int i = 0; i < N; ++i)
    cilk_spawn f();
```
Alternative: a cilk_for loop.

Code:

```c
/* empty test function */
void f() { }

void test_cilk_for()
{
    cilk_for (int i = 0; i < N; ++i)
        f();
}
```

PPA analysis:

The parallelism is about 2000 (with default grain size).
- The parallelism is high.
- As we saw earlier, this kind of loop yields good performance and speedup.
Lessons learned

- `cilk_for()` is different from `for(...)` `cilk_spawn`.
- The span of `for(...)` `cilk_spawn` is $\Omega(N)$.
- For simple flat loops, `cilk_for()` is generally preferable because it has higher parallelism.
- (However, `for(...)` `cilk_spawn` might be better for recursively nested loops.)
- Use Cilkview to measure the parallelism of your program.
Example 3: Vector addition

Code:

```c
const int N = 50 * 1000 * 1000;

double A[N], B[N], C[N];

void vector_add()
{
    cilk_for (int i = 0; i < N; ++i)
        A[i] = B[i] + C[i];
}
```

Expectations:

- Cilkview says that the parallelism is 68,377.
- This will work great!
Speedup of \texttt{vector\_add()}
Bandwidth of the memory system

A typical machine: AMD Phenom 920 (Feb. 2009).

<table>
<thead>
<tr>
<th>Cache level</th>
<th>daxpy bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>19.6 GB/s per core</td>
</tr>
<tr>
<td>L2</td>
<td>18.3 GB/s per core</td>
</tr>
<tr>
<td>L3</td>
<td>13.8 GB/s shared</td>
</tr>
<tr>
<td>DRAM</td>
<td>7.1 GB/s shared</td>
</tr>
</tbody>
</table>

**daxpy**: \(x[i] = a \times x[i] + y[i]\), double precision.

The memory bottleneck:
- A single core can generally saturate most of the memory hierarchy.
- Multiple cores that access memory will conflict and slow each other down.
How do you determine if memory is a bottleneck?

Hard problem:
- No general solution.
- Requires guesswork.

Two useful techniques:
- Use a profiler such as the Intel VTune.
  - Interpreting the output is nontrivial.
  - No sensitivity analysis.
- Perturb the environment to understand the effect of the CPU and memory speeds upon the program speed.
How to perturb the environment

- Overclock/underclock the processor, e.g. using the power controls.
  - If the program runs at the same speed on a slower processor, then the memory is (probably) a bottleneck.
- Overclock/underclock the DRAM from the BIOS.
  - If the program runs at the same speed on a slower DRAM, then the memory is not a bottleneck.
- Add spurious work to your program while keeping the memory accesses constant.
- Run $P$ independent copies of the serial program concurrently.
  - If they slow each other down then memory is probably a bottleneck.
Perturbing `vector_add()`

```c
const int N = 50 * 1000 * 1000;

double A[N], B[N], C[N];

void vector_add()
{
    cilk_for (int i = 0; i < N; ++i) {
        A[i] = B[i] + C[i];
        fib(5); // waste time
    }
}
```
Speedup of perturbed `vector_add()`
Interpreting the perturbed results

The memory is a bottleneck:

- A little extra work (\texttt{fib(5)}) keeps 8 cores busy. A little more extra work (\texttt{fib(10)}) keeps 16 cores busy.
- Thus, we have enough parallelism.
- The memory is \textit{probably} a bottleneck. (If the machine had a shared FPU, the FPU could also be a bottleneck.)

OK, but how do you fix it?

- \texttt{vector\_add} cannot be fixed in isolation.
- You must generally restructure your program to increase the reuse of cached data. Compare the iterative and recursive matrix multiplication from yesterday.
- (Or you can buy a newer CPU and faster memory.)
Lessons learned

- Memory is a common bottleneck.
- One way to diagnose bottlenecks is to perturb the program or the environment.
- Fixing memory bottlenecks usually requires algorithmic changes.
Example 4: Nested loops

Code:

```c
const int N = 1000 * 1000;

void inner_parallel()
{
    for (int i = 0; i < N; ++i)
        cilk_for (int j = 0; j < 4; ++j)
            fib(10); /* do some work */
}
```

Expectations:

- The inner loop does 4 things in parallel. The parallelism should be about 4.
- Cilkview says that the parallelism is 3.6.
- We should see some speedup.
"Speedup" of `inner_parallel()`
Interchanging loops

Code:

```c
const int N = 1000 * 1000;

void outer_parallel()
{
    cilk_for (int j = 0; j < 4; ++j)
        for (int i = 0; i < N; ++i)
            fib(10); /* do some work */
}
```

Expectations:

- The outer loop does 4 things in parallel. The parallelism should be about 4.
- Cilkview says that the parallelism is 4.
- Same as the previous program, which didn’t work.
Speedup of `outer_parallel()`
Parallelism vs. burdened parallelism

**Parallelism:**
The best speedup you can hope for.

**Burdened parallelism:**
Parallelism after accounting for the unavoidable migration overheads.

Depends upon:
- How well we implement the Cilk++ scheduler.
- How you express the parallelism in your program.

Cilkview prints the burdened parallelism:
- 0.29 for `inner_parallel()`, 4.0 for `outer_parallel()`.
- In a good program, parallelism and burdened parallelism are about equal.
What is the burdened parallelism?

Code:

```c
A();
cilk_spawn B();
C();
D();
cilk_sync;
E();
```

Burdened critical path:

The **burden** is $\Theta(10000)$ cycles (locks, malloc, cache warmup, reducers, etc.)
The burden in our examples

\( \Theta(N) \) spawns/syncs on the critical path (large burden):

```c
void inner_parallel()
{
    for (int i = 0; i < N; ++i)
        cilk_for (int j = 0; j < 4; ++j)
            fib(10); /* do some work */
}
```

\( \Theta(1) \) spawns/syncs on the critical path (small burden):

```c
void outer_parallel()
{
    cilk_for (int j = 0; j < 4; ++j)
        for (int i = 0; i < N; ++i)
            fib(10); /* do some work */
}
```
Lessons learned

- Insufficient parallelism yields *no speedup*; high burden yields *slowdown*.
- Many spawns but small parallelism: suspect large burden.
- Cilkview helps by printing the burdened span and parallelism.
- The burden can be interpreted as the number of spawns/syncs on the critical path.
- If the burdened parallelism and the parallelism are approximately equal, your program is ok.
Summary and notes

We have learned to identify and (when possible) address these problems:

- High overhead due to small grain size in cilk_for loops.
- Insufficient parallelism.
- Insufficient memory bandwidth.
- Insufficient burdened parallelism.
Plan

1. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join model and its implementation in Cilk
   - Detecting parallelization overheads and other issues
   - Anticipating parallelization overheads

2. GPU programming
   - The CUDA programming and memory models
   - Some principles for performance optimization
   - Anticipating parallelization overheads: plain polynomial division
   - Anticipating parallelization overheads: Euclidean Algorithm
   - An interlude: plain polynomial multiplication
   - Benchmarks and conclusions
### Pascal Triangle

[![Pascal Triangle](image_url)](image_url)

**Figure:** Pascal Triangle.
Divide and conquer: principle

Figure: Divide and conquer Taylor shift.
Divide and conquer: work, span and parallelism

- The **work** for a tableau satisfies
  \[ W_s(n) = 4W_s(n/2) + \Theta(1), \text{ thus:} \]
  \[ W_s(n) = \Theta(n^2). \]

- For a triangle region, we have
  \[ W_T(n) = 2W_T(n/2) + W_s(n/2), \text{ thus:} \]
  \[ W_T(n) = \Theta(n^2). \]

- The **span** for a tableau satisfies
  \[ S_s(n) = 3W_s(n/2) + \Theta(1), \text{ thus:} \]
  \[ S_s(n) = \Theta(n \log_2 3). \]

- For a triangle region, we have
  \[ S_T(n) = S_T(n/2) + S_s(n/2), \text{ thus:} \]
  \[ S_T(n) = \Theta(n \log_2 3). \]

- The **parallelism** is \( \Theta(n^{2-\log_2 3}) \), so roughly \( \Theta(n^{0.45}) \) which can be regarded as low parallelism.
Blocking strategy: principle

Figure: Blocking scheme in Pascal Triangle construction.
Blocking strategy: work, span and parallelism

Let $B$ be the order of a block and $n$ be the number of elements.

- The work (and the span) for each block is $\Theta(B^2)$.
- The number of bands (or diagonal rows) is $n/B$. Thus, the number of blocks is $\Theta((n/B)^2)$.
- The work for computing the Pascal Triangle is $\Theta(n^2)$.
- While the span is $\Theta(Bn)$
- Therefore the parallelism $\Theta(n/B)$ can still be regarded as low parallelism, but better than with the d’n’c scheme.

Figure: Blocking scheme in Pascal Triangle construction.
Estimating parallelization overheads

The instruction stream DAG of the blocking strategy consists of $n/B$ binary trees $T_0, T_1, \ldots, T_{n/B-1}$ such that

- $T_i$ is the instruction stream DAG of the cilk-for loop executing the $i$-th band
- each leaf of $T_i$ is connected by an edge to the root of $T_{i+1}$.

Consequently, the burdened span is

$$S_b(n) = \sum_{i=1}^{n/B} \log(i) = \log\left(\prod_{i=1}^{n/B} i\right) = \log(\Gamma\left(\frac{n}{B} + 1\right)) .$$

Using Stirling’s Formula, we deduce

$$S_b(n) \in \Theta\left(\frac{n}{B} \log\left(\frac{n}{B}\right)\right) . \quad (1)$$

Thus the burdened parallelism (that is, the ratio work to burdened span) is $\Theta(Bn/\log(n/B))$, that is sublinear in $n$, while the non-burdened parallelism is $\Theta(n/B)$.
Summary and notes

Burdened parallelism

- Parallelism after accounting for parallelization overheads (thread management, costs of scheduling, etc.) The burdensed parallelism is estimated as the ratio work to burdened span.
- The burdensed span is defined as the maximum number of spawns/syncs on a critical path times the cost for a cilk_spawn (cilk_sync) taken as 15,000 cycles.

Impact in practice: example for the Pascal Triangle

- Consider executing one band after another, where for each band all $B \times B$ blocks are executed concurrently.
- The non-burdened span is in $\Theta(B^2n/B) = \Theta(n/B)$.
- While the burdensed span is

$$S_b(n) = \sum_{i=1}^{n/B} \log(i) = \log(\prod_{i=1}^{n/B} i) = \log(\Gamma(\frac{n}{B} + 1)) \in \Theta\left(\frac{n}{B} \log\left(\frac{n}{B}\right)\right).$$
Plan

1 Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join model and its implementation in Cilk
   - Detecting parallelization overheads and other issues
   - Anticipating parallelization overheads

2 GPU programming
   - The CUDA programming and memory models
   - Some principles for performance optimization
   - Anticipating parallelization overheads: plain polynomial division
   - Anticipating parallelization overheads: Euclidean Algorithm
   - An interlude: plain polynomial multiplication
   - Benchmarks and conclusions
Heterogeneous programming (1/3)

- A CUDA program is a serial program with parallel kernels, all in C.
- The serial C code executes in a **host** (= CPU) thread
- The parallel kernel C code executes in many **device** threads across multiple GPU processing elements, called **streaming processors** (SP).
Thus, the parallel code (kernel) is launched and executed on a device by many threads.

Threads are grouped into thread blocks.

One kernel is executed at a time on the device.

Many threads execute each kernel.
Heterogeneous programming (3/3)

- The parallel code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in **thread and block ID variables** are used to map each thread to a specific data tile (see next slide).

- Thus, each thread executes the same code on different data based on its thread and block ID.
Example: increment array elements (1/2)

Increment N-element vector \( a \) by scalar \( b \)

Let's assume \( N=16 \), \( \text{blockDim}=4 \)  

\[
\text{int idx} = \text{blockDim.x} \times \text{blockIdx.x} + \text{threadIdx.x};
\]

\[
\begin{align*}
\text{blockIdx.x} &= 0 \\
\text{blockDim.x} &= 4 \\
\text{threadIdx.x} &= 0,1,2,3 \\
\text{idx} &= 0,1,2,3 \\
\text{blockIdx.x} &= 1 \\
\text{blockDim.x} &= 4 \\
\text{threadIdx.x} &= 0,1,2,3 \\
\text{idx} &= 4,5,6,7 \\
\text{blockIdx.x} &= 2 \\
\text{blockDim.x} &= 4 \\
\text{threadIdx.x} &= 0,1,2,3 \\
\text{idx} &= 8,9,10,11 \\
\text{blockIdx.x} &= 3 \\
\text{blockDim.x} &= 4 \\
\text{threadIdx.x} &= 0,1,2,3 \\
\text{idx} &= 12,13,14,15
\end{align*}
\]

See our example number 4 in /usr/local/cs4402/examples/4
Example: increment array elements (2/2)

**CPU program**

```c
void increment_cpu(float *a, float b, int N)  
{  
    for (int idx = 0; idx<N; idx++)  
        a[idx] = a[idx] + b;  
}  

void main()  
{  
    ......  
    increment_cpu(a, b, N);  
}  
```

**CUDA program**

```c
__global__ void increment_gpu(float *a, float b, int N)  
{  
    int idx = blockIdx.x * blockDim.x + threadIdx.x;  
    if( idx < N)  
        a[idx] = a[idx] + b;  
}  

void main()  
{  
    ......  
    dim3 dimBlock (blocksize);  
    dim3 dimGrid( ceil( N / (float)blocksize) );  
    increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);  
}  
```
A **Thread block** is a group of threads that can:
- Synchronize their execution
- Communicate via shared memory

Within a grid, **thread blocks can run in any order**:
- Concurrently or sequentially
- Facilitates scaling of the same code across many devices
Thread blocks (2/2)

- Thus, within a grid, any possible interleaving of blocks must be valid.

- Thread blocks **may coordinate but not synchronize**
  - they may share pointers
  - they should not share locks (this can easily deadlock).

- The fact that thread blocks cannot synchronize gives **scalability**:
  - A kernel scales across any number of parallel cores

- However, within a thread block, threads may synchronize with barriers.

- That is, threads wait at the barrier until **all** threads in the **same block** reach the barrier.
Blocks run on multiprocessors
Streaming processors and multiprocessors
Hardware multithreading

- **Hardware allocates resources to blocks:**
  - blocks need: thread slots, registers, shared memory
  - blocks don’t run until resources are available
- **Hardware schedules threads:**
  - threads have their own registers
  - any thread not waiting for something can run
  - context switching is free every cycle
- **Hardware relies on threads to hide latency:**
  - thus high parallelism is necessary for performance.
SIMT thread execution

- At each clock cycle, a multiprocessor executes the same instruction on a group of threads called a **warp**
  - The number of threads in a warp is the **warp size** (32 on G80)
  - A half-warp is the first or second half of a warp.
- Within a warp, threads
  - share instruction fetch/dispatch
  - some become inactive when code path diverges
  - hardware automatically handles divergence
- **Warps are the primitive unit of scheduling:**
  - each active block is split into warps in a well-defined way
  - threads within a warp are executed physically in parallel while warps and blocks are executed logically in parallel.
Plan

1 Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join model and its implementation in Cilk
   - Detecting parallelization overheads and other issues
   - Anticipating parallelization overheads

2 GPU programming
   - The CUDA programming and memory models
   - Some principles for performance optimization
   - Anticipating parallelization overheads: plain polynomial division
   - Anticipating parallelization overheads: Euclidean Algorithm
   - An interlude: plain polynomial multiplication
   - Benchmarks and conclusions
Four principles

- Expose as much parallelism as possible
- Optimize memory usage for maximum bandwidth
- Maximize occupancy to hide latency
- Optimize instruction usage for maximum throughput
Expose Parallelism

- Structure algorithm to maximize independent parallelism

- If threads of same block need to communicate, use shared memory and __syncthreads()

- If threads of different blocks need to communicate, use global memory and split computation into multiple kernels

- Recall that there is no synchronization mechanism between blocks

- High parallelism is especially important to hide memory latency by overlapping memory accesses with computation

- Take advantage of asynchronous kernel launches by overlapping CPU computations with kernel execution.
Optimize Memory Usage: Basic Strategies

- Processing data is cheaper than moving it around:
  - Especially for GPUs as they devote many more transistors to ALUs than memory

- Basic strategies:
  - Maximize use of low-latency, high-bandwidth memory
  - Optimize memory access patterns to maximize bandwidth
  - Leverage parallelism to hide memory latency by overlapping memory accesses with computation as much as possible
  - Write kernels with high arithmetic intensity (ratio of arithmetic operations to memory transactions)
  - Sometimes recompute data rather than cache it
Minimize CPU $\leftarrow \rightarrow$ GPU Data Transfers

- CPU $\leftarrow \rightarrow$ GPU memory bandwidth much lower than GPU memory bandwidth

- Minimize CPU $\leftarrow \rightarrow$ GPU data transfers by moving more code from CPU to GPU
  - Even if sometimes that means running kernels with low parallelism computations
  - Intermediate data structures can be allocated, operated on, and deallocated without ever copying them to CPU memory

- Group data transfers: One large transfer much better than many small ones.
Optimize Memory Access Patterns

- Effective bandwidth can vary by an order of magnitude depending on access pattern:
  - Global memory is not cached on G8x.
  - Global memory has High latency instructions: 400-600 clock cycles
  - Shared memory has low latency: a few clock cycles

- Optimize access patterns to get:
  - Coalesced global memory accesses
  - Shared memory accesses with no or few bank conflicts and
  - to avoid partition camping.
A Common Programming Strategy

1. Partition data into subsets that fit into shared memory
2. Handle each data subset with one thread block
3. Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism.
4. Perform the computation on the subset from shared memory.
5. Copy the result from shared memory back to global memory.
A Common Programming Strategy

Partition data into subsets that fit into shared memory
A Common Programming Strategy

Handle each data subset with one thread block
A Common Programming Strategy

Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism.
A Common Programming Strategy

Perform the computation on the subset from shared memory.
Copy the result from shared memory back to global memory.
A Common Programming Strategy

- Carefully partition data according to access patterns
- If read only, use `__constant__` memory (fast)
- for read/write access within a tile, use `__shared__` memory (fast)
- for read/write scalar access within a thread, use registers (fast)
- R/W inputs/results `cudaMalloc`ed, use global memory (slow)
Plan

1. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join model and its implementation in Cilk
   - Detecting parallelization overheads and other issues
   - Anticipating parallelization overheads

2. GPU programming
   - The CUDA programming and memory models
   - Some principles for performance optimization
   - Anticipating parallelization overheads: plain polynomial division
   - Anticipating parallelization overheads: Euclidean Algorithm
   - An interlude: plain polynomial multiplication
   - Benchmarks and conclusions
Plain univariate polynomial division

**Input:** univariate polynomials \( a = \sum_{0}^{m} a_i x^i \) and \( b = \sum_{0}^{n} b_i x^i \) in \( R[x] \) with respective degrees \( m \) and \( n \) such that \( m \geq n \geq 0 \) and \( b_n \) is a unit.

**Output:** the quotient \( q \) and the remainder \( r \) of \( a \) w.r.t. \( b \). Hence \( a = bq + r \) and \( \text{deg} \ r < n \).

\[
\begin{align*}
r &:= a \\
\text{for } i &= m - n, m - n - 1, \ldots, 0 \text{ repeat} \\
\text{if } \text{deg} \ r &= n + i \text{ then} \\
q_i &:= \text{leadingCoefficient}(r) / b_n \\
r &:= r - q_i x^i b \\
\text{else } q_i &:= 0 \\
q &:= \sum_{0}^{m-n} q_i x^i \\
\text{return } (q, r)
\end{align*}
\]
Plain univariate polynomial division on the GPU

Notations
Consider two univariate polynomials over a finite field

\[ a = a_m x^m + \cdots + a_1 x + a_0 \quad \text{and} \quad b = b_n x^n + \cdots + b_1 x + b_0, \quad \text{with} \quad m \geq n, \]

stored in arrays \( A \) and \( B \) such that \( A[i] \) (resp. \( B[i] \)) contains \( a_i \) (resp. \( b_i \)).

Objective, challenges and key idea
- Our aim is to implement the plain division of \( A \) by \( B \).
- Thus we have no other choices than parallelizing each division step.
- Each thread block must know the leading coefficient of the current remainder at the beginning of each division step.
- In order to minimize data transfer (and synchronization) we will let each thread block work on several consecutive division steps without synchronizing.
- For this to be possible, each thread block within a kernel computes the the leading coefficient of the current remainder.
**Division: data mapping**

**Key idea, more precisely**

- Let $s > 1$ be an integer.
- After each kernel call either the current intermediate remainder has become zero or its degree has decreased at least by $s$.
- Moreover, each kernel call performs at most $s$ division steps.

**Data mapping**

- The grid and each thread block are 1-D.
- Initially, the $k$-th thread block has a window on
  - the $s$ leading terms of $A$ and the $s$ leading terms of $B$.
  - $2s$ consecutive coefficients $a_{m-(2k+1)s} \cdots a_{m-(2k+3)s+1}$ of $A$
  - $3s$ consecutive coefficients $b_{n-(2k+1)s+g_0} \cdots a_{m-(2k+4)s+1+g_0}$ of $B$ where $g_0 = \deg(a) - \deg(b)$.

\[
A = \begin{bmatrix}
    a_m & \cdots & a_{m-s+1} \\
    a_{m-s} & \cdots & a_{m-3s+1} \\
    \vdots & & \vdots \\
    \cdots & \cdots & a_0 & 0 & \cdots & 0
\end{bmatrix}
\]
\[
B = \begin{bmatrix}
    b_n & \cdots & b_{n-s+1} \\
    b_{n-s+g_0} & \cdots & a_{m-4s+1+g_0} \\
    \vdots & & \vdots \\
    \cdots & \cdots & b_0 & 0 & \cdots & 0
\end{bmatrix}
\]
Key idea, more precisely

- Let $s > 1$ be an integer.
- After each kernel call either the current intermediate remainder has become zero or its degree has decreased at least by $s$.
- Moreover, each kernel call performs at most $s$ division steps.
Division: algorithm

Data mapping (recall)

\[
A = \begin{vmatrix} a_m \cdots a_{m-s+1} \\ \vdots \end{vmatrix} \quad \begin{vmatrix} a_{m-s} \cdots a_{m-3s+1} \\ \vdots \end{vmatrix} \quad \begin{vmatrix} \cdots \end{vmatrix} \quad \begin{vmatrix} \cdots \end{vmatrix} \\
B = \begin{vmatrix} b_n \cdots b_{n-s+1} \\ \vdots \end{vmatrix} \quad \begin{vmatrix} b_n-s+g_0 \cdots a_{m-4s+1+g_0} \\ \vdots \end{vmatrix} \quad \begin{vmatrix} \cdots \end{vmatrix} \quad \begin{vmatrix} \cdots \end{vmatrix} 
\]

Algorithm

- Let \( a^{(j)} \) be the \( j \)-th intermediate remainder, with \( a^{(0)} = a \).
- Let \( \ell_j = \text{lc}(a^{(j)}) \), \( g_j = \deg(a_j) - \deg(b) \) and \( a^{(j)}_i = \text{coeff}(a^{(j)}, x^i) \). We have
  \[
  a_i^{(j)} = a_i^{(j-1)} - \frac{\ell_{j-1}}{b_n} b_i - g_j.
  \]
- Each thread block computes \( \ell_j \) until the degree of the intermediate remainder has decreased at least by \( s \).
- During the execution of a kernel call \( A[i] \) stores successively \( a_i^{(0)}, a_i^{(1)}, \ldots \), that is, the coefficient of degree \( i \) of the intermediate remainder.
- At the end of a kernel call, the \( k \)-th thread block has the coefficients of the current remainder in degree \( m - (2k + 1)s, \ldots, m - (2k + 3)s + 1 \).
Division: complexity analysis

Work

- The number of kernel calls is at most $\lceil \frac{m-n}{s} \rceil$
- The number of thread blocks per kernel call is $\lceil \frac{n}{2s} \rceil$
- The number of arithmetic operations per thread block is at most $6s^2$
- Thus the work is in $O((n - m)n)$, as expected.
- However, there is an increase of work w.r.t. a serial division by a constant factor in order to keep synchronization overhead low.

Span

- The number of kernel calls is at most $\lceil \frac{m-n}{s} \rceil$
- The number of division steps per kernel call is at most $s$
- Thus the span is in $O(m - n)$, as expected.

In practice

On our GPU card $s = 2^9$. 
Plan

1. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join model and its implementation in Cilk
   - Detecting parallelization overheads and other issues
   - Anticipating parallelization overheads

2. GPU programming
   - The CUDA programming and memory models
   - Some principles for performance optimization
   - Anticipating parallelization overheads: plain polynomial division
   - Anticipating parallelization overheads: Euclidean Algorithm
   - An interlude: plain polynomial multiplication
   - Benchmarks and conclusions
GCD: setting

Notations
Consider two univariate polynomials over a finite field

\[ a = a_m x^m + \cdots + a_1 x + a_0 \quad \text{and} \quad b = b_n x^n + \cdots + b_1 x + b_0, \quad \text{with} \quad m \geq n, \]

stored in arrays \( A \) and \( B \) such that \( A[i] \) (resp. \( B[i] \)) contains \( a_i \) (resp. \( b_i \)).

Objective, challenges and key idea
- Our aim is to implement the Euclidean Algorithm for computing \( \text{gcd}(A, B) \).
- Again, we have no other choices than parallelizing each division step.
- Once again, in order to minimize data transfer (and granularity) we will let each thread block work on several consecutive division steps without synchronizing.
- For this to be possible, each thread block within a kernel computes the leading coefficients of the current pairs.
GCD: algorithm and data mapping

Algorithm

- Let $s > 1$ be an integer.
- Each kernel call replaces $(A, B)$ by a GCD preserving pair $(A', B')$ such that \( \max(\deg(A), \deg(B)) - \max(\deg(A'), \deg(B')) \geq s \)
- Moreover, each kernel call performs at most $s$ division steps.
- If initially $| \deg(A) - \deg(B) | \geq s$ we can call simply use the kernel for $s$ division step with a fixed divisor.
- If initially $| \deg(A) - \deg(B) | \geq s$, we need to increase the window on $A$ (or $B$) since the divisor may change after each division step.

Data mapping for $| \deg(A) - \deg(B) | \geq s$

- The grid and each thread block are 1-D.
- Initially, the $k$-th thread block has a window on
  - the $s$ leading terms of $A$ and the $s$ leading terms of $B$.
  - $3s$ consecutive coefficients $a_m-(2k+1)s \cdots a_m-(2k+4)s+1$ of $A$
  - $3s$ consecutive coefficients $b_{n-(2k+1)s+g_0} \cdots a_m-(2k+4)s+1+g_0$ of $B$ where $g_0 = \deg(a) - \deg(b) \geq 0$.

\[
A = \begin{vmatrix}
    a_m \cdots a_{m-s+1} \\
    a_{m-s} \cdots a_{m-4s+1} \\
    \vdots \\
    \cdots a_0 \cdots 0
\end{vmatrix}
\]
\[
B = \begin{vmatrix}
    b_n \cdots b_{n-s+1} \\
    b_{n-s+g_0} \cdots a_{m-4s+1+g_0} \\
    \vdots \\
    \cdots b_0 \cdots 0
\end{vmatrix}
\]
GCD: complexity analysis

Work
- Assume \( m \geq n \).
- The number of kernel calls is at most \( \lceil \frac{m}{s} \rceil \).
- The number of thread blocks per kernel call is at most \( \lceil \frac{m}{2s} \rceil \).
- The number of arithmetic operations per thread block is at most \( 6s^2 \).
- Thus the work is in \( O(m^2) \), as expected.
- However, there is an increase of work w.r.t. a serial GCD computation by a constant factor in order to keep the granularity low.
- Moreover, there is an increase of memory consumption w.r.t. the GPU division computation by a constant factor due to the case where the degree gap is small.

Span
- The number of kernel calls is at most \( \lceil \frac{m}{s} \rceil \).
- The number of division steps per kernel call is at most \( s \).
- Thus the span is in \( O(m) \), which as good as in the case of systolic arrays (H.T. Kung & C.E. Leiserson, 1974).

In practice
On our GPU card \( s = 2^8 \).
Plan

1. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join model and its implementation in Cilk
   - Detecting parallelization overheads and other issues
   - Anticipating parallelization overheads

2. GPU programming
   - The CUDA programming and memory models
   - Some principles for performance optimization
   - Anticipating parallelization overheads: plain polynomial division
   - Anticipating parallelization overheads: Euclidean Algorithm
   - An interlude: plain polynomial multiplication
   - Benchmarks and conclusions
Plain multiplication: setting

Notations
Consider two univariate polynomials over a finite field

\[ a = a_m x^m + \cdots + a_1 x + a_0 \quad \text{and} \quad b = b_n x^n + \cdots + b_1 x + b_0, \quad \text{with} \quad m \geq n, \]

stored in arrays \( A \) and \( B \) such that \( A[i] \) (resp. \( B[i] \)) contains \( a_i \) (resp. \( b_i \)). For simplicity we assume \( n = m \) in what follows.

Objective, challenges and key idea

- Computing \( A \times B \) using the plain algorithm the way we learned it in primary school.
- First we construct all the terms concurrently, thus essentially in time \( O(1) \).
- Secondly, we sum the terms concurrently, using a parallel reduction, thus essentially in time \( O(\log(n)) \).
Plain multiplication: data mapping (1/4)

Given two polynomials of degree 19, thus of (dense) size 20:

\[ A := X X X X X X X X X X X X X X X X X X X X \]
\[ B := X X X X X X X X X X X X X X X X X X X X \]

the multiplication space is

```
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X
```
Let \( r > 1 \) be an integer dividing \( n + 1 \). We partition the multiplication space horizontally \( (n + 1)/r \) bands.
Plain multiplication: data mapping (3/4)

Let $t > 1$ be an integer dividing $n + 1$. We partition each horizontal band into $(n + 1)/t$ polygons
Plain multiplication: data mapping (4/4)

The leftmost and rightmost polygons are padded with zeros such each polygon becomes a rectangle.
Plain multiplication: in colors!
Plain multiplication: algorithm

Multiplication phase

- We have now $\frac{n+1}{r} \frac{n+1}{t}$ rectangular tiles.
- Each tile requires $t + r - 1$ coefficients from $A$ and $r$ from $B$.
- Each tile is computed by a thread block, where each thread is in charge of $s$ consecutive columns. Clearly, the grid is 2-D.
- The $i$-th horizontal band is mapped to an array $C_i$ of size $2n - 1$.
- Each thread on the $i$-th horizontal band sums the elements of a column in the appropriate coefficient of $C_i$.

Addition phase

- The vectors $C_0, C_1, \ldots, C_w$, with $w = n/r$, are added such that terms of the same degree are added together.
- For $i = 0, 2, 4, \ldots$, the vectors $C_i$ and $C_{i+1}$ are added into $C_i$.
- For $i = 0, 4, 8, \ldots$, the vectors $C_i$ and $C_{i+2}$ are added into $C_i$.
- Etc.
- In $\Theta(\log(w))$ parallel steps all vectors $C_0, C_1, \ldots, C_w$ are added together yielding the final result.
Plain multiplication: complexity analysis

Multiplication phase

- We have $\frac{n+1}{r} \frac{n+1}{t}$ thread blocks.
- Each thread block has a work of $2r(t + r - 1)$ and a span $sr$.
- Thus the work is essentially $\Theta(n^2)$ while the parallelism is $\Theta(n^2/sr)$.
- For our GPU card $r = t = 2 \times 9$ and $s \in \{4, 6\}$.

Addition phase

- The work is essentially is $2nw$, with $w = n/r$.
- The span is $\log(w)$.
- Thus the overall parallelism is $\Theta(n^2/\log(w))$
Plan

1. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join model and its implementation in Cilk
   - Detecting parallelization overheads and other issues
   - Anticipating parallelization overheads

2. GPU programming
   - The CUDA programming and memory models
   - Some principles for performance optimization
   - Anticipating parallelization overheads: plain polynomial division
   - Anticipating parallelization overheads: Euclidean Algorithm
   - An interlude: plain polynomial multiplication
   - Benchmarks and conclusions
The plain multiplication algorithm on the GPU

<table>
<thead>
<tr>
<th>degree</th>
<th>GPU Plain multiplication</th>
<th>GPU FFT based multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{10}$</td>
<td>0.00049</td>
<td>0.0044136</td>
</tr>
<tr>
<td>$2^{11}$</td>
<td>0.0009</td>
<td>0.004642912</td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>0.0032</td>
<td>0.00543696</td>
</tr>
<tr>
<td>$2^{13}$</td>
<td>0.01</td>
<td>0.00543696</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>0.045</td>
<td>0.00709072</td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>0.26</td>
<td>0.006796512</td>
</tr>
</tbody>
</table>

- The GPU card is Tesla 2050 (two years old) and the CPU is an i7 (same desktop).
- All computations (plain ones and fast ones) of univariate products are done on the GPU.
- Both codes are highly optimized.
## GPU plain division vs NTL serial fast division (part 1)

<table>
<thead>
<tr>
<th>m</th>
<th>n</th>
<th>CUDA</th>
<th>NTL (9001)</th>
<th>NTL (7)</th>
<th>NTL (469762049)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>500</td>
<td>0.0013567</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>500</td>
<td>0.00394246</td>
<td>0.004</td>
<td>0.004</td>
<td>0.008</td>
</tr>
<tr>
<td>2000</td>
<td>1500</td>
<td>0.00135683</td>
<td>0.004</td>
<td>0</td>
<td>0.004</td>
</tr>
<tr>
<td>3000</td>
<td>500</td>
<td>0.00652643</td>
<td>0.004</td>
<td>0.004</td>
<td>0.008</td>
</tr>
<tr>
<td>3000</td>
<td>1500</td>
<td>0.00394755</td>
<td>0.008</td>
<td>0.004</td>
<td>0.008</td>
</tr>
<tr>
<td>3000</td>
<td>2500</td>
<td>0.00135741</td>
<td>0</td>
<td>0.004</td>
<td>0</td>
</tr>
<tr>
<td>4000</td>
<td>500</td>
<td>0.00911792</td>
<td>0.004</td>
<td>0.004</td>
<td>0.012</td>
</tr>
<tr>
<td>4000</td>
<td>1500</td>
<td>0.00653302</td>
<td>0.012001</td>
<td>0.016001</td>
<td>0.020001</td>
</tr>
<tr>
<td>4000</td>
<td>2500</td>
<td>0.00394336</td>
<td>0.008</td>
<td>0.008</td>
<td>0.012</td>
</tr>
<tr>
<td>4000</td>
<td>3500</td>
<td>0.00135872</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
</tr>
<tr>
<td>5000</td>
<td>500</td>
<td>0.0117174</td>
<td>0.008</td>
<td>0.004</td>
<td>0.012001</td>
</tr>
<tr>
<td>5000</td>
<td>1500</td>
<td>0.00911808</td>
<td>0.012</td>
<td>0.012</td>
<td>0.020001</td>
</tr>
<tr>
<td>5000</td>
<td>2500</td>
<td>0.00653037</td>
<td>0.016001</td>
<td>0.016001</td>
<td>0.024001</td>
</tr>
<tr>
<td>5000</td>
<td>3500</td>
<td>0.00394688</td>
<td>0.008</td>
<td>0.008</td>
<td>0.012001</td>
</tr>
<tr>
<td>5000</td>
<td>4500</td>
<td>0.00135827</td>
<td>0</td>
<td>0.004</td>
<td>0.004</td>
</tr>
<tr>
<td>m</td>
<td>n</td>
<td>CUDA</td>
<td>NTL (9001)</td>
<td>NTL (7)</td>
<td>NTL (469762049)</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>----------</td>
<td>------------</td>
<td>--------</td>
<td>-----------------</td>
</tr>
<tr>
<td>6000</td>
<td>2000</td>
<td>0.0103908</td>
<td>0.012001</td>
<td>0.012</td>
<td>0.016001</td>
</tr>
<tr>
<td>6000</td>
<td>4000</td>
<td>0.00523638</td>
<td>0.008</td>
<td>0.008</td>
<td>0.012001</td>
</tr>
<tr>
<td>6000</td>
<td>5000</td>
<td>0.00264752</td>
<td>0.004</td>
<td>0.004</td>
<td>0.008</td>
</tr>
<tr>
<td>7000</td>
<td>2000</td>
<td>0.0129791</td>
<td>0.012001</td>
<td>0.012001</td>
<td>0.020001</td>
</tr>
<tr>
<td>7000</td>
<td>4000</td>
<td>0.00781933</td>
<td>0.016001</td>
<td>0.012</td>
<td>0.024001</td>
</tr>
<tr>
<td>7000</td>
<td>6000</td>
<td>0.00264899</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
</tr>
<tr>
<td>8000</td>
<td>2000</td>
<td>0.0155647</td>
<td>0.016001</td>
<td>0.016001</td>
<td>0.024001</td>
</tr>
<tr>
<td>8000</td>
<td>4500</td>
<td>0.00910278</td>
<td>0.016001</td>
<td>0.016001</td>
<td>0.024002</td>
</tr>
<tr>
<td>8000</td>
<td>7000</td>
<td>0.00265875</td>
<td>0.004</td>
<td>0.004</td>
<td>0.008</td>
</tr>
<tr>
<td>9000</td>
<td>2000</td>
<td>0.01815</td>
<td>0.016001</td>
<td>0.016001</td>
<td>0.024001</td>
</tr>
<tr>
<td>9000</td>
<td>5000</td>
<td>0.0103951</td>
<td>0.016001</td>
<td>0.016001</td>
<td>0.004</td>
</tr>
<tr>
<td>9000</td>
<td>8000</td>
<td>0.00265123</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
</tr>
<tr>
<td>10000</td>
<td>2000</td>
<td>0.0207235</td>
<td>0.016001</td>
<td>0.016001</td>
<td>0.024002</td>
</tr>
<tr>
<td>10000</td>
<td>5500</td>
<td>0.0116883</td>
<td>0.028001</td>
<td>0.036002</td>
<td>0.052003</td>
</tr>
<tr>
<td>10000</td>
<td>9000</td>
<td>0.00264813</td>
<td>0.004</td>
<td>0.004</td>
<td>0.008001</td>
</tr>
</tbody>
</table>
# GPU plain GCD vs NTL serial fast GCD (part 1)

<table>
<thead>
<tr>
<th>m</th>
<th>n</th>
<th>CUDA</th>
<th>NTL (9001)</th>
<th>NTL (7)</th>
<th>NTL (469762049)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>500</td>
<td>0.0104428</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
</tr>
<tr>
<td>2000</td>
<td>500</td>
<td>0.0178581</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
</tr>
<tr>
<td>2000</td>
<td>1500</td>
<td>0.0247671</td>
<td>0.024001</td>
<td>0.024001</td>
<td>0.024001</td>
</tr>
<tr>
<td>3000</td>
<td>500</td>
<td>0.0232968</td>
<td>0.004</td>
<td>0.008</td>
<td>0.008</td>
</tr>
<tr>
<td>3000</td>
<td>1500</td>
<td>0.0321629</td>
<td>0.028002</td>
<td>0.024001</td>
<td>0.036002</td>
</tr>
<tr>
<td>3000</td>
<td>2500</td>
<td>0.0393252</td>
<td>0.056003</td>
<td>0.060003</td>
<td>0.056003</td>
</tr>
<tr>
<td>4000</td>
<td>500</td>
<td>0.0319935</td>
<td>0.008</td>
<td>0.004</td>
<td>0.012001</td>
</tr>
<tr>
<td>4000</td>
<td>1500</td>
<td>0.0384393</td>
<td>0.036002</td>
<td>0.032002</td>
<td>0.044003</td>
</tr>
<tr>
<td>4000</td>
<td>2500</td>
<td>0.0474535</td>
<td>0.056003</td>
<td>0.056003</td>
<td>0.056003</td>
</tr>
<tr>
<td>4000</td>
<td>3500</td>
<td>0.0533327</td>
<td>0.060003</td>
<td>0.072004</td>
<td>0.072005</td>
</tr>
<tr>
<td>5000</td>
<td>500</td>
<td>0.0375882</td>
<td>0.008</td>
<td>0.008</td>
<td>0.016001</td>
</tr>
<tr>
<td>5000</td>
<td>1500</td>
<td>0.047392</td>
<td>0.036002</td>
<td>0.032002</td>
<td>0.040002</td>
</tr>
<tr>
<td>5000</td>
<td>2500</td>
<td>0.0523071</td>
<td>0.052003</td>
<td>0.052003</td>
<td>0.056004</td>
</tr>
<tr>
<td>5000</td>
<td>3500</td>
<td>0.060711</td>
<td>0.092006</td>
<td>0.096006</td>
<td>0.100006</td>
</tr>
<tr>
<td>5000</td>
<td>4500</td>
<td>0.0692999</td>
<td>0.112007</td>
<td>0.096006</td>
<td>0.116007</td>
</tr>
</tbody>
</table>
## GPU plain GCD vs NTL serial fast GCD (part 2)

<table>
<thead>
<tr>
<th>m</th>
<th>n</th>
<th>CUDA</th>
<th>NTL (9001)</th>
<th>NTL (7)</th>
<th>NTL (469762049)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6000</td>
<td>2000</td>
<td>0.0400133</td>
<td>0.040002</td>
<td>0.044002</td>
<td>0.048003</td>
</tr>
<tr>
<td>6000</td>
<td>4000</td>
<td>0.0518378</td>
<td>0.096006</td>
<td>0.104006</td>
<td>0.128008</td>
</tr>
<tr>
<td>6000</td>
<td>5000</td>
<td>0.0565259</td>
<td>0.124008</td>
<td>0.136008</td>
<td>0.148009</td>
</tr>
<tr>
<td>7000</td>
<td>2000</td>
<td>0.0456542</td>
<td>0.044003</td>
<td>0.040002</td>
<td>0.052003</td>
</tr>
<tr>
<td>7000</td>
<td>4000</td>
<td>0.0568928</td>
<td>0.116007</td>
<td>0.108007</td>
<td>0.116007</td>
</tr>
<tr>
<td>7000</td>
<td>6000</td>
<td>0.0666983</td>
<td>0.152009</td>
<td>0.152009</td>
<td>0.16801</td>
</tr>
<tr>
<td>8000</td>
<td>2000</td>
<td>0.0527113</td>
<td>0.044002</td>
<td>0.044003</td>
<td>0.040003</td>
</tr>
<tr>
<td>8000</td>
<td>4500</td>
<td>0.0627832</td>
<td>0.136008</td>
<td>0.112006</td>
<td>0.128008</td>
</tr>
<tr>
<td>8000</td>
<td>7000</td>
<td>0.0767803</td>
<td>0.156009</td>
<td>0.16001</td>
<td>0.200013</td>
</tr>
<tr>
<td>9000</td>
<td>2000</td>
<td>0.0578016</td>
<td>0.048003</td>
<td>0.048003</td>
<td>0.052003</td>
</tr>
<tr>
<td>9000</td>
<td>5000</td>
<td>0.0714158</td>
<td>0.140009</td>
<td>0.152009</td>
<td>0.168011</td>
</tr>
<tr>
<td>9000</td>
<td>8000</td>
<td>0.0876858</td>
<td>0.16801</td>
<td>0.216013</td>
<td>0.264017</td>
</tr>
<tr>
<td>10000</td>
<td>2000</td>
<td>0.0635697</td>
<td>0.16801</td>
<td>0.040002</td>
<td>0.05200</td>
</tr>
<tr>
<td>10000</td>
<td>5500</td>
<td>0.0796034</td>
<td>0.180011</td>
<td>0.16001</td>
<td>0.200013</td>
</tr>
<tr>
<td>10000</td>
<td>9000</td>
<td>0.0976259</td>
<td>0.232014</td>
<td>0.244015</td>
<td>0.264017</td>
</tr>
</tbody>
</table>
### GPU plain GCD

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n$</th>
<th>CUDA with $s = 512$</th>
<th>CUDA with $s = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>500</td>
<td>0.010</td>
<td>0.024</td>
</tr>
<tr>
<td>2000</td>
<td>1500</td>
<td>0.024</td>
<td>0.058</td>
</tr>
<tr>
<td>3000</td>
<td>2500</td>
<td>0.039</td>
<td>0.108</td>
</tr>
<tr>
<td>4000</td>
<td>3500</td>
<td>0.053</td>
<td>0.158</td>
</tr>
<tr>
<td>5000</td>
<td>4500</td>
<td>0.069</td>
<td>0.203</td>
</tr>
<tr>
<td>6000</td>
<td>5000</td>
<td>0.056</td>
<td>0.235</td>
</tr>
<tr>
<td>7000</td>
<td>6000</td>
<td>0.066</td>
<td>0.282</td>
</tr>
<tr>
<td>8000</td>
<td>7000</td>
<td>0.076</td>
<td>0.324</td>
</tr>
<tr>
<td>9000</td>
<td>8000</td>
<td>0.087</td>
<td>0.367</td>
</tr>
<tr>
<td>10000</td>
<td>9000</td>
<td>0.097</td>
<td>0.411</td>
</tr>
</tbody>
</table>

- We observe that **trick with leading coefficients** is necessary to achieve our 3 to 4 time speedup factor w.r.t NTL.
- In other words, controlling parallelization overheads (due here to synchronization) is necessary to reach a positive result.
Summary and notes

- The GPU plain division running times grows (essentially) linearly with $m - n$, as for systolic arrays.
Summary and notes

- The GPU plain division running times grows (essentially) linearly with $m - n$, as for systolic arrays.
- The GPU plain GCD running times grows (essentially) linearly with $\max(n, m)$, as for systolic arrays.
Summary and notes

- The GPU plain division running times grows (essentially) linearly with $m - n$, as for **systolic arrays**.
- The GPU plain GCD running times grows (essentially) linearly with $\max(n, m)$, as for systolic arrays.
- This degree range of 5,000 to 10,000 for which the GPU plain arithmetic is clearly better is what we need for solving challenging polynomial systems (but not yet for cryptosystems).
Summary and notes

- The GPU plain division running times grows (essentially) linearly with \( m - n \), as for systolic arrays.
- The GPU plain GCD running times grows (essentially) linearly with \( \max(n, m) \), as for systolic arrays.
- This degree range of 5,000 to 10,000 for which the GPU plain arithmetic is clearly better is what we need for solving challenging polynomial systems (but not yet for cryptosystems).
- With more recent and more powerful cards the gap between the GPU plain arithmetic and the FFT-based arithmetic should greatly increase in favor of the former.
Summary and notes

- The GPU plain division running times grows (essentially) linearly with $m - n$, as for *systolic arrays*.
- The GPU plain GCD running times grows (essentially) linearly with $\max(n, m)$, as for systolic arrays.
- This degree range of 5,000 to 10,000 for which the GPU plain arithmetic is clearly better is what we need for solving challenging polynomial systems (but not yet for cryptosystems).
- With more recent and more powerful cards the gap between the GPU plain arithmetic and the FFT-based arithmetic should greatly increase in favor of the former.
- With CUDA, parallel for-loop overheads are low by design. By minimizing synchronization overheads, we had a (parallel) quadratic algorithm beating a (serial) quasi-linear one.
The GPU plain division running times grows (essentially) linearly with \( m - n \), as for **systolic arrays**.

The GPU plain GCD running times grows (essentially) linearly with \( \max(n, m) \), as for systolic arrays.

This degree range of 5,000 to 10,000 for which the GPU plain arithmetic is clearly better is what we need for solving challenging polynomial systems (but not yet for cryptosystems).

With more recent and more powerful cards the gap between the GPU plain arithmetic and the FFT-based arithmetic should greatly increase in favor of the former.

With CUDA, parallel for-loop overheads are low by design. By minimizing synchronization overheads, we had a (parallel) quadratic algorithm beating a (serial) quasi-linear one.

Did I use a hammer to beat a fly? Not really.
Summary and notes

- The GPU plain division running times grows (essentially) linearly with $m - n$, as for systolic arrays.
- The GPU plain GCD running times grows (essentially) linearly with $\max(n, m)$, as for systolic arrays.
- This degree range of 5,000 to 10,000 for which the GPU plain arithmetic is clearly better is what we need for solving challenging polynomial systems (but not yet for cryptosystems).
- With more recent and more powerful cards the gap between the GPU plain arithmetic and the FFT-based arithmetic should greatly increase in favor of the former.
- With CUDA, parallel for-loop overheads are low by design. By minimizing synchronization overheads, we had a (parallel) quadratic algorithm beating a (serial) quasi-linear one.
- Did I use a harmer to beat a fly? Not really.
- Algebraic complexity is clearly no longer the main complexity measure!
Summary and notes

- The GPU plain division running times grows (essentially) linearly with $m - n$, as for systolic arrays.
- The GPU plain GCD running times grows (essentially) linearly with $\max(n, m)$, as for systolic arrays.
- This degree range of 5,000 to 10,000 for which the GPU plain arithmetic is clearly better is what we need for solving challenging polynomial systems (but not yet for cryptosystems).
- With more recent and more powerful cards the gap between the GPU plain arithmetic and the FFT-based arithmetic should greatly increase in favor of the former.
- With CUDA, parallel for-loop overheads are low by design. By minimizing synchronization overheads, we had a (parallel) quadratic algorithm beating a (serial) quasi-linear one.
- Did I use a harmer to beat a fly? Not really.
- Algebraic complexity is clearly no longer the main complexity measure!
- Yes, GPUs bring supercomputing to the masses!